Panoramica sulle tecnologie e sugli strumenti per la programmazione parallela
(I parte)

Giampaolo Bottoni, Maurizio Cremonesi
CILEA, Segrate

Abstract
Con la progressiva diffusione di ambienti di calcolo multiprocessore anche di piccola taglia, diventa sempre più importante conoscere quali sono le problematiche di base e le tecnologie disponibili per il calcolo parallelo. Il CILEA offre una larga gamma di soluzioni che consentono sia di sfruttare i 36 processori del cluster attualmente operativo (un HP V2500 e due HP N4000) sia di realizzare programmi portabili ovunque sia possibile disporre da due ad N processori (con N anche superiore al centinaio) per realizzare calcoli scientifici di elevato impegno computazionale.

In questo articolo si desidera inquadrare la attuale situazione e disponibilità di strumenti per la programmazione sui calcolatori CILEA dedicati al calcolo scientifico in un contesto più generale. Pur essendo da "lustri" disponibili ambienti per il calcolo scientifico ad alte prestazioni, solo da poco tempo (due, tre anni) si ha la sensazione che la situazione si vada "standardizzando" dal punto di vista di una offerta omogenea di strumenti per il calcolo parallelo. Il fatto concreto che, a parere degli scriventi, dovrebbe determinare la saldatura tra due mondi tuttora piuttosto lontani, quello del supercalcolo fattibile in siti "privilegiati" tipo il CILEA e quello delle applicazioni di tutti i giorni, dovrebbe essere la disponibilità di personal computers biprocessori, quadriprocessori etc., in gergo: a molte vie. Anche se la diffusione di tali "micro-supercalcolatori" sarà determinata da ragioni totalmente diverse da quella di soddisfare esigenze di calcolo scientifico è indubbio che la disponibilità generalizzata di macchine per cui risulta vantaggioso saper pilotare non una singola CPU ma due o quattro o più, dovrebbe favorire la diffusione delle conoscenze nella programmazione parallela (per l'occasione chiamata "multithread"). Naturalmente il fatto che si riducano gli ostacoli economici ad acquisire competenza e all'utilizzo del supercalcolo è condizione solo necessaria ma non sufficiente a far sì che l'uso del calcolo, più o meno intensivo, diventi pratica diffusa prima di tutto in ambito universitario e poi, per contagio, nel mondo industriale italiano. Purtroppo, perché certe cose si avverino non basta che siano auspicabili da molti punti di vista incluso quello della cultura scientifica Esistono varie altre condizioni almeno "quasi" necessarie quanto il fatto di poter essere presto a 30 cm di distanza da un PC biprocessore che non si sa usare: tra queste la stabilità e la standardizzazione degli strumenti indispensabili per realizzare programmi paralleli ovvero linguaggi, compilatori e librerie dedicate allo scopo. Qui la mescolanza tra strumenti "astratti" (linguaggi) e "concreti" (compilatori e librerie) è voluta e intende sottolineare il fatto che, per un buon risultato globale non basta purtroppo, che qualche gruppo di "ben pensanti" abbia proposto anche il più azzeccato standard di programmazione di questo mondo se poi nessuna software house si prende la briga di realizzare un compilatore aderente a quel bellissimo standard.
Attualmente però, a nostro parere, linguaggi e compilatori sono ormai adeguati per sostenere questa piccola campagna per la diffusione delle competenze di programmazione parallela.
Un punto piuttosto rilevante per sostenere questa tesi è quello di dimostrare che, al limite, per far calcolo parallelo bastano già gli strumenti base di tutti i giorni ovvero un compilatore (ma senza bachi!!!) aderente allo standard Fortran 90 più una sola direttiva di pa-

rallelizzazione (un commento che permetta di imporre che quanto segue deve essere attuato non da una CPU ma da tutte quelle volute). Volendo dare a questo articolo anche un certo grado di utilità pratica si cercherà di dare consistenza operativa all' illustrazione dei principi base e dei vari aspetti applicativi dei principali metodi disponibili al CILEA per realizzare un eseguibile parallelo. Immaginando inoltre che il lettore sia persona con nozioni di programmazione ma senza esperienza specifica di programmazione parallela verranno segnalate anche alcune risorse disponibili su internet per approfondire gli argomenti qui accennati.

I punti base del calcolo parallelo

Anche se per gli addetti ai lavori sono considerazioni ovvie è opportuno ribadire qui quali sono i termini della questione genericamente catalogata come calcolo parallelo – linguaggi e strumenti per… Perché la questione abbia interesse per qualcuno questo qualcuno (ricercatore universitario o industriale):

- Deve avere a che fare con un problema che richieda una mole di calcolo rilevante, misurata sulla scala delle possibilità di un calcolatore *normale* in quel momento.
- Deve avere reali possibilità di intervento sulla procedura di calcolo attuabile. Dunque accesso al codice e capacità e libertà di modifiche e ristrutturazioni anche radicali.
- Deve essere interessato alla durata dei calcoli da svolgere (considerando un calcolo infattibile per limitazioni dell' hardware equivalente ad un calcolo di durata infinita)

Soddisfatte le premesse e tenendo presente dunque che l'ottimizzazione delle prestazioni è l'esigenza di fondo del calcolo parallelo, la situazione ideale sarebbe quella in cui tutto avvenisse miracolisticamente, grazie a strumenti automatici di ristrutturazione di algoritmi e di programmi. Questa situazione ideale si realizza solo raramente perché il problema è per sua natura complesso e richiede una capacità di analisi e di ristrutturazione che difficilmente uno strumento automatico possiede.

E' intuitivo il concetto di collaborazione tra vari processori, in buona parte analogo a quello di lavoro di gruppo da parte di un certo numero di esseri umani. Quello che è normalmente poco evidente, se non facendo mente locale, è la conseguenza delle altissime velocità di esecuzione delle CPU attualmente in commercio. Il punto cruciale ovvero l'aspetto determinante per decidere se ha senso attivare più di una sola CPU dedicata al uno specifico problema è la necessità e frequenza dei punti di sincronizzazione nel lavoro di gruppo. Si consideri, per similitudine, un gruppo di persone che lavorano assieme; le riunioni sono indispensabili ed utili ma costituiscono un fattore di rallentamento delle attività se alcuni componenti del gruppo sono costretti a sospendere il lavoro in attesa delle decisioni della riunione. Analogamente un calcolo a molti processori non ha senso (ovvero non richiede un tempo ridotto rispetto al lavoro solitario) se i granuli di lavoro assegnati a ciascun processore sono eseguibili troppo velocemente rispetto ai tempi di assegnazione dei granuli stessi o se i granuli di lavoro hanno durate troppo differenti da processore a processore: un numero elevato di processori si troverebbe ad aver completato il suo compito molto prima degli altri e resterebbe dunque inattivo fino alla successiva fase di sincronizzazione. Ma il dire che un granulo di lavoro è piccolo dipende dalla velocità della singola CPU e con CPU operanti ormai attorno ad 1 GHz si possono fare 1000 moltiplicazioni in un solo milionesimo di secondo !

Dunque, per capire il problema: allo stato della attuale tecnologia il calcolo parallelo ha senso se la nostra procedura comporta molto di più che l'effettuazione di un miliardo di moltiplicazioni e se è possibile distribuire compiti di sufficiente consistenza, più di un migliaio di moltiplicazioni, a processori distinti che debbono essere obbligati a ri- sincronizzarsi e a scambiarsi risultati con frequenza la più bassa possibile, ragionevolmente molto più lenta di una sincronizzazione ogni microsecondo.

Come ottenere la cooperazione

Nel corso degli anni sono stati proposti vari schemi per attuare la cooperazione tra i processori. Ogni schema ha cercato e dovuto fare dei compromessi per conciliare le difficoltà derivanti dalla tipologia delle macchine usate con le esigenze umane, del programmatore molto spesso non propenso ad essere condizionato dalle peculiarità dell'hardware. A favore della affermazione di approcci "universali" ha gioca-

to una sorte di selezione darwiniana che ha portato all'estinzione vari tipi di calcolatori troppo di nicchia e all'affermazione di calcolatori relativamente meno costosi perché basati su componenti giustificati da ragioni commerciali enormemente più ampie.
I principali schemi adottati attualmente sono raggruppabili nelle seguenti categorie:

- Metodi basati sull'utilizzo di direttive di assegnazione delle istruzioni. Si tratta dell'approccio forse di più antica data, applicato al nascere dei primi calcolatori vettoriali
- Metodi basati sull'utilizzo di direttive di assegnazione dei dati (che implicano, ma come conseguenza, la suddivisione dei compiti tra processori)
- Matodi basati sullo scambio esplicito di messaggi tra processori che eseguono programmi più o meno simili usando dati differenti. La diversità del codice eseguito tra un processore e l'altro può essere molto rilevante anche se, per ragioni di leggibilità e manutenzione si tende a realizzare programmi con poche diversità di compiti tra processori.
- Metodi basati sulla assegnazione esplicita di compiti specifici a "filoni di lavoro" distinti, solitamente detti threads.
- Metodi basati sull'estensione del linguaggio (promozione di scalari a vettori, di vettori a matrici, etc. ossia utilizzo di indici aggiuntivi per esprimere la molteplicità delle immagini attive in parallelo: è il così detto metodo delle CoArray)

Sostanziamente (fa eccezione il metodo delle CoArray), al CILEA è possibile realizzare e applicare programmi basati su tutti gli approcci ora elencati e le questioni di attualità riguardano aspetti apparentemente non di primaria importanza ma in realtà piuttosto critici per la diffusione del calcolo parallelo quali la standardizzazione delle direttive, in corso ma ancora incompleta e l'aderenza integrale agli standard da parte dei compilatori, precompilatori, ottimizzatori, ora disponibili.
Il fatto che esista una certa diversità di soluzioni per affrontare praticamente lo stesso problema è forse l'indicazione della relativa giovinezza del problema stesso e del fatto che a differenza di ciò che è successo a livello di hardware non ha avuto ancora il tempo di verificarsi una "selezione naturale" a proposito delle tecnologie applicate. Il fatto è che probabilmente, a favore della diversificazione, hanno giocato un ruolo importante diversi fattori legati al contesto applicativo:

- Esigenze di conservazione di programmi preesistenti concepiti inizialmente solo per ambienti di calcolo monoprocessore
- Vincoli culturali degli utilizzatori delle tecnologie ossia di quella particolare tipologia di ricercatore-programmatore motivato a volte solo marginalmente ad interessarsi di problematiche di ottimizzazione informatica "suis generis" quale è la parallelizzazione di un codice.
- Conflitti "filosofico-accademici" sulle caratteristiche e peculiarità di linguaggi di programmazione destinati ad applicazioni di nicchia.
- Competizione tra i produttori di hardware per favorire la diffusione degli schemi funzionalmente più adatti alle caratteristiche dei propri prodotti.

Dal punto di vista di chi deve adottare uno o l'altro approccio la situazione ha come lato negativo solo il fatto che si è obbligati a fare delle scelte, messo che questa possibilità di scelta sia reale. I metodi basati sullo scambio di messaggi risultano normalmente di applicabilità più gravosa rispetto a quelli che richiedono l'uso di direttive. Tali metodi però consentono di rendere portabile il codice praticamente nella totalità degli ambienti paralleli oggi disponibili ossia in ambienti sia a memoria distribuita quale potrebbe essere un gruppo di calcolatori connessi in rete, sia in ambienti a memoria fisica condivisa tipo un PC biprocessore, sia a memoria solo virtualmente condivisa, tipo i server adibiti al calcolo intensivo al CILEA, un HP V2500 e due HP N4000.
I metodi basati sull'uso di direttive sono di applicabilità più facile ma comportano rischi di degrado delle prestazioni connessi alla imperizia nell'utilizzo senza cognizione di causa. Il rischio è piuttosto elevato soprattutto a proposito delle direttive di assegnazione dei dati (direttive HPF == High Performance Fortran) poiché tali direttive, in ambienti fisicamente distribuiti, possono implicare un trasferimento anche imponente di dati da un calcolatore all'altro con conseguente fortissimo aumento dei tempi di esecuzione. Per quanto riguarda la programmazione di threads il van-

taggio è sostanzialmente quello di poter applicare esperienza di programmazione acquisita nella realizzazione di programmi applicativi nell' ambito del calcolo non scientifico. Nell'applicazione in sistemi a memoria condivisa la programmazione a threads tende ad essere eclissata dalla disponibilità di direttive di parallelizzazione e quindi ha preso poco piede ma potrebbe acquisire un certo rilievo in futuro per favorire la portabilità tra ambienti con diverso sistema operativo (Windows / Unix). La disponibilità di tanti paradigmi di programmazione largamente equivalenti in ambienti a memoria condivisa e a parità di competenza nel loro utilizzo anche se non completamente equiparabili dal punto di vista della facilità d'uso e della portabilità è un indizio dell'importanza dell' "amichevolezza" della programmazione parallela e consente di passare al capitolo successivo di questa panoramica intitolato provocatoriamente....

Fortran questo sconosciuto

Le basi su cui è possibile sostenere le proprie preferenze per un linguaggio piuttosto che per un altro sono di carattere solo parzialmente oggettive. Qualunque linguaggio di programmazione è valido se chi lo usa "sente" che risponde bene alle proprie esigenze e basa il proprio giudizio su confronti di produttività con utilizzatori di linguaggi diversi: quello che sarebbe veramente auspicabile è il dialogo, l'interoperatività tra componenti realizzati nel modo più diverso.

Tuttavia è indubbio che un linguaggio di programmazione è un mosaico di "invenzioni" ed è un organismo in più o meno veloce evoluzione, capace di inglobare le idee nuove. Un nuovo linguaggio avrà il pregio di poter fare tesoro di tutte le "invenzioni" pregresse e sarà dotato di un alto livello di coerenza interna; non sarà però necessariamente più funzionale di un linguaggio precedente se non risponderà esattamente alle stesse esigenze per cui il linguaggio precedente è stato formulato. Una di queste esigenze, per quanto concerne il Fortran, è stata quella di essere, per l'epoca, il più vicino possibile alle tradizioni della grafia matematica e il più "digeribile" per quella famosa categoria di gente che pensa che un codice di calcolo scientifico sia una meccanica traduzione di formule.

Se dunque siamo favorevoli a che ogni studente di discipline scientifiche sia obbligato a programmare in Java (così impara!) siamo anche (almeno al 50%) dell'opinione che la conoscenza del "relativamente facile" Fortran sia sufficiente alla realizzazione di buoni programmi di calcolo parallelo purché esista la consapevolezza dei reali motivi per cui il passaggio (da quasi un decennio ormai) dal Fortran 77 al Fortran 90 abbia segnato un deciso progresso positivo.

Dunque se Fortran vuol dire pressoché totale ignoranza su come strutturare un insieme di procedure che oltre alle pure formule contengono dati, e tanti, per farne un programma di qualità almeno semi-professionale, allora meglio Java. Ma il Fortran basta ed avanza se si è capito a cosa servono i moduli ossia, in parole povere, specie di scatoloni in cui mettere dichiarazioni di tipi di dati derivati, dichiarazioni di vettori e matrici e funzioni e procedure studiate specificamente per operare su quei particolari dati.

In altre parole sembra piuttosto irragionevole che uno che affermi di conoscere il Fortran 90 si dichiari favorevole ad un linguaggio ad oggetti quale è Java senza aver capito che un modulo rappresenta, almeno a livello embrionale, un oggetto con almeno alcune delle caratteristiche per cui ha ragione di essere la programmazione ad oggetti.

Il nesso logico di queste considerazioni con il tema dell'articolo va cercato nel fatto che, avendo consapevolezza del problema da risolvere ossia come organizzare un programma di calcolo in modo da essere ben strutturato per l'esecuzione parallela, la soluzione può essere trovata anche solo ricorrendo ad istruzioni di puro Fortran 90 associate ad accorgimenti ispirati dalla nozione stessa di oggetto propria della programmazione ad oggetti. In ultima analisi, dunque, si cercherà di mostrare come la disponibilità di una direttiva di attivazione di thread o almeno la disponibilità di una funzione che permetta l'attivazione esplicita di un thread consenta la realizzazione di un programma di calcolo parallelo in qualunque ambiente (Unix o Windows) in cui abbia senso eseguire in modo parallelo un programma (tranne che per ragioni didattiche, sarà ovviamente indispensabile disporre di almeno due reali CPU per poter ottenere una effettiva riduzione dei tempi di esecuzione).

Calcolo parallelo con moduli

Quanto segue può essere di utilità non solo a titolo di illustrazione dell'essenza della problematica insita nella realizzazione di un programma parallelo ma anche come spunto per la realizzazione di programmi paralleli in ambienti in cui si desideri ricorrere direttamente alle funzioni messe a disposizione dal sistema operativo per l'attivazione di thread (tipo la `CreateThread` in Windows). Il programma, qui descritto per sommi capi è stato sperimentato sul HP V2500 a 20 processori, utilizzando la versione 2.4 del compilatore HP F90.

Il punto di partenza sia sintetizzabile, almeno per fissare le idee, in questi termini: so come dovrebbe essere realizzato un buon programma parallelo ossia:

- Strutturato in modo che sia per lo meno facile, se non immediata, la realizzazione di una versione a scambio di messaggi (realizzabile in concreto usando una libreria MPI reperibile in commercio o anche di pubblico dominio). Dunque un programma con caratteristiche di buona portabilità anche in ambienti distribuiti.
- Efficiente ossia impostato in modo da raggiungere facilmente una granularità a grana grossa ovvero in modo che i compiti assegnati ad ogni thread siano adeguatamente impegnativi per fare in modo che gli istanti di sincronizzazione nei quali tutti i thread debbono aver completato il lavoro loro assegnato prima di proseguire, siano il meno frequenti possibile.
- Facilmente gestibile o almeno gestibile quanto un normale programma sequenziale e dunque facilmente utilizzabile come un normale programma sequenziale.

Ho a disposizione solo un buon compilatore F90 capace di accettare chiamate alle API di … (un nome a caso) Windows 98 ma voglio poter essere in grado di portare velocemente il mio programma sui superpotenti calcolatori del CILEA pur lasciandomi realmente aperta la porta all'esecuzione in ambiente a reale parallelismo con Win. NT o 2000.

Avendo qualche nozione di programmazione ad oggetti so che una caratteristica essenziale per l'incapsulamento dei dati è il fatto che i componenti di una classe sono per default privati ossia inaccessibili al di fuori del file di definizione della classe stessa. In Fortran 90 le variabili dichiarate all'interno di un modulo come pure le procedure (function o subroutine) e gli eventuali tipi derivati sono pubblici ma è possibile, con l'istruzione PRIVATE capovolgere completamente la situazione ovvero rendere invisibili tutti gli oggetti contenuti in un modulo tranne quelli dichiarati esplicitamente pubblici. Questo fatto mi consente di usare procedure con lo stesso nome in moduli diversi. Infatti anche se userò entrambi i moduli contemporaneamente il compilatore non potrà accorgersi dell'omonimia delle procedure perché l'istruzione PRIVATE renderà reciprocamente invisibili i contenuti dei due moduli.

Superato il problema dell'utilizzo duplicato di subroutine o function aventi lo stesso nome nasce il problema del riuscire ad attivarle facendo in modo che ciascuna di esse venga eseguita da un diverso thread ossia, avendo a disposizione più di una singola CPU, in modo realmente parallelo usando tutte le CPU del mio PC usato come ambiente di sviluppo.

Per fare questo basterà personalizzare ciascuno dei moduli che inscatolano le subroutine omonime realizzando una subroutine che funga da "pseudo_main" di quel modulo e che pertanto venga dichiarata esplicitamente PUBLIC in modo da poter essere assegnata al thread. Lo pseudo main di ciascun modulo essendo posto fisicamente dentro il suo modulo scatolone avrà accesso a tutte le aree dati e a tutte le subroutine incluse nel proprio modulo e dunque potrà operare con subroutine identiche ma utilizzanti aree di memoria diverse e attivate da thread diversi.

Ecco, dunque realizzato un vero programma parallelo funzionante in ambiente Windows con semplici CALL alla function `CreateThread` e in ambiente Unix, sui calcolatori del CILEA, con una singola direttiva che forzi la parallelizzazione.

Analizziamo ora in dettaglio quanto ora esposto a grandi linee e cercando di supercommentare i sorgenti per non disorientare troppo chi non è pratico delle diavolerie dell'F90.

```
module modpubblico ! (1)
    !  Tutti vedono questi dati
    type,public::memproc
        integer::chisono
        real(8),dimension(:),pointer::w
    end type memproc
    integer::iterazioni=100
```

```
    integre,dimension(64)::numcaso
    real(8)::piccolissimo=1.d+40
    !___altre dichiarazioni___!
contains
    subroutine imposta_pubblico()
        integer:: k
        do k=1,size(numcaso,1)
            numcaso(k)=k*3
        end do
    end subroutine imposta_pubblico
    !___altre procedure interne___!
end module modpubblico
```

Tabella 1

Innanzi tutto vediamo come potrebbe essere un modulo visibile globalmente a tutte le procedure che ne dovessero far uso.
Nella tabella 1 viene mostrato un esempio tipico. Il modulo si chiama "modpubblico" e contiene le definizioni di vari dati tra cui il vettore numcaso di 64 elementi. Il modulo contiene anche una subroutine di nome "imposta_pubblico" che consente di inizializzare nel modo voluto il vettore numcaso: come tutte le subroutine contenute in un modulo, la "imposta_pubblico" è visibile solo dalle procedure che usano il modulo "pubblico".
Nel modulo pubblico c'è un esempio di definizione di un tipo di dato personale che, per l'occasione, è stato battezzato `type(memproc)`. Un oggetto dichiarato di `type(memproc)` contiene, in questo esempio, un indice che permette di sapere quale deve essere il numero convenzionale del thread abilitato a gestire i dati dell'oggetto ed un esempio di vettore di dati reali che, avendo l'attributo pointer, potrà venire fisicamente allocato in base all'esigenza. Dunque l'occupazione di memoria fisica potrà variare da uno all'altro oggetto di `type( memproc)` ossia i vari thread potranno usare quantità diverse di memoria.
Vediamo ora come è fatto il tipico modulo chiave, il "pilastro" su cui poggia tutta la logica di questa gestione parallela:

```
module parallelo_p7 ! (2)
    use modpubblico                !_b1
    private                        !_b2
    public main_p7                 !_b3
    integer,parameter::nomen=7     !_b4
    include "aliasmem.htm"         !_b5
contains
    include "aliaslib.htm"         !_b6
    subroutine main_p7(cose)       !_b7
       type(memproc) ::cose
       !___ varie istruzioni ___!
       Call alias_sub1(cose)
       Call alias_sub2(cose)
       !___ varie istruzioni ___!
       return
    end subroutine main_p7
end module parallelo_p7
```

Tabella 2

In tabella 2 viene riportato un modulo chiamato "`parallelo_p7`" dove il 7 sta a segnalare che dovrebbero essere realizzati diversi moduli esattamente identici tra loro tranne che per la cifra: un modulo per ciascuno dei thread che si intende usare e dunque, pensando che ogni thread vada eseguito da un diverso processore, tanti moduli fatto così quante sono le CPU a propria disposizione.
Si noti che mentre il modulo `modpubblico` ha un contenuto solo esemplificativo, il modulo `parallelo_p7` è esattamente (a parte dettagli marginali) quello che deve essere scritto per qualsiasi programma reale che si voglia parallelizzare nel modo ora suggerito. In altre parole, avendo a disposizione 8 CPU dovremo materialmente scrivere 8 moduli costituiti da quella ventina di righe del prototipo qui riportato.
Commentiamo le istruzioni chiave di `parallelo_p7`. L'istruzione !_b1, la `use`, permette di accedere a dati e alle dichiarazioni fatte in `modpubblico`. L'istruzione !_b2 è di importanza cruciale: tutti i dati e tutte le subroutine diventano invisibili anche se una subroutine fa una `use` di `parallelo_p7`. Dunque non debbo preoccuparmi se un altro modulo dichiara variabili e procedure usando nomi identici a quelli usati in `parallelo_p7`. L'istruzione !_b3 dichiara che main_p7 è un nome visibile. Si tratta del nome della subroutine pubblica tramite cui intendo attivare le altre subroutine, rese altrimenti inaccessibili, del modulo `parallelo_p7`. L'istruzione !_b4, il parametro `nomen` consente alle subroutine contenute nel modulo di sapere il numero identificativo del modulo stesso. Ogni modulo parallelo dovrà avere questa dichiarazione ma ovviamente il valore del parametro dovrà essere diverso da modulo a modulo. L' istruzione !_b5 è una classica istruzione di `include`. Tutti i moduli includono gli stessi file (in questo caso `aliasmem.htm`) che danno luogo a variabili omonime ma fisicamente distinte. In questo modo la gestione del mio programma è semplice ed identica a quella di un normale programma per singolo processore: ho un solo file di di-

chiarazioni da trattare indipendentemente dal numero di CPU che voglio usare. L'istruzione !_b6 è una seconda `include`. Mentre la prima includeva il file aliasmem.htm con le dichiarazione delle variabili, questa serve per includere la libreria di subroutine invisibili dall'esterno ossia il file `aliaslib.htm`.
Incidentalmente si noti il fatto che l'estensione di entrambi i file inclusi è htm. È molto facile fare in modo che un file contenente istruzioni Fortran in formato libero sia anche un file contenente tag HTML e dunque sia leggibile su internet. Per evitare interferenze tra i linguaggi basta rendere commenti Fortran i tag HTML ed eventualmente nascondere al browser il simbolo di commento, il punto esclamativo, ponendolo all'interno dei tag HTML. Con questi accorgimenti è possibile documentare i sorgenti Fortran facilitandone la leggibilità con tutti gli accorgimenti consentiti dall'HTML.
L'istruzione !_b7 rappresenta l'inizio dell'unica subroutine pubblica del modulo `parallelo_p7` ovvero il main del settimo thread.
Si sottolinea qui l'importanza di scrivere questa subroutine nel modo più semplice possibile poiché tutta la sostanza del programma, in realtà, può essere scritta dentro una qualsiasi subroutine della libreria contenuta nel file `aliaslib.htm`. Mentre però ci sarà un unico file `aliaslib.htm` il numero di moduli paralleli potrebbe essere piuttosto elevato per cui bisogna ridurre l'esigenza di doverli modificare tutti magari solo per cambiare un parametro o il valore di inizializzazione di una variabile.
Si noti che `main_p7` riceve in argomento un elemento dichiarato `type(memproc)` che si suppone sia stato opportunamente inizializzato con il numero di riconoscimento del thread. In questo modo ogni subroutine della libreria del file aliaslib.htm (che è identica per tutti i moduli paralleli) può sapere il numero del thread per cui lavora semplicemente controllando la variabile `cose%chisono` (si noti che in Fortran 90 non si usa il carattere "." ma il carattere "%" per collegare il nome di un componente al nome dell'oggetto che lo contiene).
Consideriamo ora un esempio di file `aliaslib.htm`.

```
!<html><head><title>Lib. Aliaslib</title><!—
!--> </head><body><pre>
recursive subroutine alias_sub1(o)   !_c1
    implicit none
    type(memproc) ::o                !_c2
    !__varie istruzioni__!
    if(o%chimono .ne. nomen) stop    !_c3
    !__varie istruzioni__!
    Return
end subroutine alias_sub1
!__varie altre subroutine e function__!
!</pre></body></html>
```

Tabella 3

Le prime due righe e l'ultima sono dei commenti Fortran ma servono per la strutturazione HTML del file; si osservi in particolare il fatto che il punto esclamativo che commenta, dal punto di vista Fortran, la seconda riga si trova all'interno di un tag HTML, che in particolare è un tag di commento HTML, per cui viene ignorato dall'HTML stesso.
L'istruzione !_c1 rappresenta una tipica prima istruzione di subroutine che viene dichiarata ricorsiva anche se nessuna istruzione interna alla subroutine fa una call alla subroutine stessa. In realtà, anche se nel contesto delle subroutine di libreria parallela la dichiarazione di `recursive` è inutile essa viene qui fatta per sottolineare l'importanza che tutte le subroutine non incluse in questo file ma richiamate dalle subroutine di questo file siano dichiarate ricorsive. Normalmente le variabili interne di una subroutine occupano fisicamente una unica posizione in memoria; dichiarando la ricorsività si ottiene che le variabili interne vengano allocate dinamicamente ad ogni chiamata e questo evita disastrosi conflitti di accesso nel caso in cui due thread attivino contemporaneamente la stessa subroutine.
L'istruzione !_c2 dichiara il tipo dell'argomento della variabile o. Si usa qui un nome piuttosto insolito ("o" iniziale di omnia) per ragioni di leggibilità: è più leggibile `o%chisono` piuttosto che usare nomi lunghi del tipo `mieidati%chisono`. Il tipo derivato `memproc` ha il compito di contenere dati distinti, diversi da thread a thread di variabili che hanno lo stesso scopo. Si tratta in sostanza di un metodo alternativo a quello dei moduli duplicati per fare in modo che ogni thread disponga di una sua memoria privata. Rispetto all'uso di moduli duplicati suggerita qui come la via principale per attuare il parallelismo, il sistema detto il metodo degli "oggettoni" è forse più elegante perché il vettore di oggettoni, con tanti elementi quanti sono i thread usati è configurabile automaticamente senza dunque l'onere della duplicazione semimanuale dei

moduli paralleli. Lo svantaggio è però il fatto che ogni subroutine che usa le componenti di un oggettone deve usare nomi del tipo o%j, o%k etc. invece che, semplicemente, j,k, etc. In altre parole le istruzioni della subroutine vanno riscritte una per una e la presenza della scritta "o%" davanti a ciascun nome danneggia alquanto la leggibilità del sorgente.
In !_c3 viene dato un esempio di uso sia di una componente dell'oggettone sia di variabile parametrica. Il compilatore che non ammette dichiarazioni implicite (`implicit none`) riconosce la variabile `nomen` perché trova tale variabile dichiarata nel modulo parallelo che funge da "scatolone" della subroutine `alias_sub1`. Analogamente, per via indiretta, il compilatore scopre quale deve essere la struttura del tipo derivato `memproc`. Dunque la subroutine `alias_sub1`, compilata a sé stante, senza che venga inclusa in un modulo, produce diagnostici di errore grave. Volendo conservare la possibilità di usare la subroutine in modo tradizionale si potrà comunque eliminare i diagnostici inserendo una istruzione di `use` di un modulo opportuno contenente le informazioni mancanti.
Non resta che analizzare la struttura del programma principale che ha lo scopo di inizializzare i moduli paralleli e di attivare i thread.
Per funzionare in ambiente HP Unix sul V2500 del CILEA dovrebbe essere ad esempio come mostrato qui:

```
Program test
    use modpubblico         !_t1
    use parallelo_p1        !_t2
    !__le use di tutti i moduli paralleli __!
    implicit none
    integer,parameter::quanti_th=4
    integer::np
    type(memproc), &
        dimension(quanti_th)::cose   !_t3
    !__inizializzazioni varie __!
!$DIR LOOP_PARALLEL
    do np=1,quanti_th
        select case(np)
        case(1)
            call main_p1(cose(1))
        case(2)
            call main_p2(cose(2))
        !__altri case, uno per thread__!
        end select
    end do
    !__calcoli sequenziali__!
    stop
End program test
```

Tabella 4

Si noti innanzitutto il fatto che, benché ogni modulo parallelo faccia uso del modulo `modpubblico`, il program deve a sua volta fare una `use` a tale modulo poiché il compilatore non vede il contenuto dei vari moduli paralleli a causa dell'istruzione `PRIVATE` in ciascuno di essi. Ogni modulo parallelo (vedi istruzione !_t2) deve essere esplicitamente usato e quindi il programma compilato è applicabile a non più di `quanti_th` thread paralleli.
Nella istruzione !_t3 viene dichiarato un vettore di "oggettoni" che vanno inizializzati in modo opportuno in base alle esigenze del problema. Il fulcro del programma è rappresentato dal ciclo di do che è preceduto dalla direttiva `LOOP_PARALLEL`. Tale direttiva è obbligatoria perché il compilatore, di norma non parallelizza cicli di do contenenti chiamate a subroutine esterne; con questa direttiva chi programma si prende la responsabilità di assicurare che le subroutine non interferiscono reciprocamente poiché operano su memorie fisicamente distinte. In funzione dell'indice del do, tramite la select case, viene richiamata l'unica subroutine pubblica di ciascun modulo parallelo. Per concludere non resta che elencare come le varie subroutine contenute nei moduli paralleli, dichiarate private e quindi inaccessibili possano attingere i dati di ingresso e fornire i risultati:

- Tramite vettori dichiarati nel modulo `modpubblico`. Ogni thread dovrà modificare solo alcuni degli elementi di tali vettori per cui si ha modo di attuare anche sistemi di sincronizzazione "morbida" tra i vari thread basati sulla lettura, da parte di un thread, dei dati scritti dagli altri.
- Tramite le componenti di ciascun oggettone che viene passato in argomento alla subroutine pubblica di ciascun modulo parallelo.
- Tramite ulteriori subroutine dichiarate pubbliche, contenute nei moduli paralleli, realizzate per "estrarre" valori di grandezze private. Questa soluzione comporta l'obbligo e l'onere di usare subroutine di nome diverso da modulo a modulo per evitare conflitti di nome nel programma principale.

Per concludere un cenno alle esigenze di una versione per ambiente Windows.

La funzione che attiva i thread non accetta ovviamente tipi derivati dichiarati in Fortran ma richiede di ricevere il nome della procedura che il thread deve eseguire e tale procedura ha una struttura rigida ovvero deve avere un solo argomento ovvero un integer corrispondente al numero convenzionale del thread. Si aggira questo vincolo inserendo la dichiarazione del vettore di oggettoni direttamente nel modulo `modpubblico` per cui l'intero vettore diventa (un po' pericolosamente) visibile alle subroutine dei vari moduli paralleli e si può limitare il rischio di interferenza solo facendo usare a ciascuna subroutine l'elemento corrispondente al valore del parametro `nomen`.

Automatizzazione del metodo

Quanto fin qui esposto, pur concettualmente semplice, richiede un certo lavoro manuale per essere applicato in pratica. D'altra parte è innegabile che il metodo abbia un certo interesse pratico se si pensa che si è arrivati, con pochissimi accorgimenti, a definire una metodologia di parallelizzazione che sgrava completamente l'utilizzatore persino dall'uso di semplici ed ovvie direttive e che stimola a "comportamenti virtuosi" ossia efficienti perché il programmatore, dovendo incapsulare in subroutine il lavoro di ogni thread sarà indotto a realizzare subroutine complesse, dunque "computazio-nalmente pesanti" e dunque a realizzare "grossi granuli di lavoro". Abbiamo dunque pensato di mettere a disposizione di chiunque fosse interessato all'argomento, un vero e proprio (piccolo) sistema per l'automatizzazione di questa procedura.
Si trova all'indirizzo:

`http://www.cilea.it/servizi/b/00/001/`

e consente di realizzare programmi paralleli tanto in ambiente Windows che Unix (sperimentato sul V2500 del CILEA). La parallelizzazione di un programma non viene attuata usando direttive particolari ma semplicemente suddividendo il programma in varie parti logiche riconosciute dal compilatore attraverso opportune convenzioni sul nome dei files.
Le varie parti logiche del programma *tipico* sono:

- Le procedure parallele (nome convenzionale: aliaslib.htm) attivabili solo per chiamata reciproca a cascata da una chiamata del programma principale parallelo.
- Il programma principale parallelo (nome convenzionale: aliasmain.htm) di cui esistono tante immagini identiche quanto vale il numero massimo di thread gestiti dal driver.
- La memoria locale ad ogni thread (nome convenzionale: aliasmem.htm) visibile al programma principale e alle subroutine appartenenti alla stessa immagine.
- I moduli pubblici visibili dalle memorie locali (nome convenzionale: aliaspub.htm) uno solo dei quali deve esistere obbligatoriamente perché usato di default da ogni immagine.
- Il driver del calcolo complessivo (epoche1.htm) che nasconde le direttive di parallelizzazione in ambiente Unix o le chiamate alle API Windows.

Il driver viene generato automaticamente tramite un apposito programma che richiede solo il numero massimo di thread e l'ambiente di utilizzo. Si tratta di un programma scritto anch'esso in Fortran 90 e che è facilmente personalizzabile per quanto concerne i nomi convenzionali da dare al modulo pubblico di default, ai moduli paralleli, ai nomi dei file da includere etc...
Il driver del programma parallelo viene pilotato dal main parallelo e dalle subroutine della libreria parallela secondo la seguente logica.

- Il main parallelo è una subroutine che viene attribuita a ciascun thread dal driver. Viene riattivata dal driver in base alla richiesta fatta dal main parallelo nella precedente attivazione.
- Il driver gestisce un indice di fase accessibile attraverso la function `epoca()`. Il main parallelo ha il compito di svolgere le operazioni proprie della particolare epoca e quindi di effettuare un return che rappresenta un punto di sincronizzazione per tutti i thread attivi a quell'epoca.
- L'epoca viene incrementata solo se non esiste nessuna immagine che richiede di mantenerla invariata.
- Il programma parallelo può modificare la frequenza con cui viene riattivata l'immagine attiva associata ad un dato thread, tramite la function `new_age(n)`. Se n ha valore negativo o nullo l'immagine corri-

spondente al valore posseduto dalla variabile nomen verrà riattivata alla stessa epoca. Se n ha valore positivo l'immagine viene riattivata ad un'epoca incrementata di n rispetto all'epoca in corso.

- Ad ogni epoca saranno attivi i thread delle immagini che nella fase di sincronizzazione precedente si sono prenotati per essere attivi per quell'epoca. Si può dunque far variare da epoca ad epoca il numero di immagini attive.
- Se la `new_age` riceve in argomento il valore data dal parametro requiem_aeterna il thread corrispondente a quella immagine non verrà più riattivato ossia verrà considerato morto e l'esecuzione termina quando tutte la immagini si sono suicidate.

In conclusione, disponendo di poche funzioni e variabili di ambiente ovvero: nomen, num_threads, epoca(), new_age(), si possono realizzare programmi SPMD operativi non solo dove è disponibile un reale ambiente parallelo ma anche in qualunque ambiente sequenziale.

Calcolo parallelo con direttive

L'uso di direttive per il compilatore rappresenta uno dei più antichi metodi per mantenere un controllo su quanto fatto in modo automatico, sulla base di quanto desumibile dalla analisi del sorgente, impedendo ad esempio operazioni corrette sul piano teorico ovvero consentite dall'algoritmo implementato ma magari fortemente sconsigliate a livello di prestazione del calcolo ovvero tali da allungare anche sensibilmente la durata del calcolo nonostante l'impiego intensivo di risorse costose (le CPU utilizzate). La necessità di usare direttive esplicite si è progressivamente ridotta con l'aumentare della qualità dell'analisi automatica svolta dal compilatore ma resta il fatto che esistono situazioni in cui oggettivamente è impossibile desumere dalla pura analisi del sorgente l'esistenza di interferenze o dipendenze tra le varie fasi dell'attività tali da impedire l'esecuzione dei calcoli in modo completamente indipendente da un granulo di lavoro all'altro. Tipico e importante esempio di necessità di uso di direttive è il seguente:

```
!$dir loop_parallel
do k=1,n
     call esegui_questo(k)
end do
```

Il compilatore in assenza della direttiva che forza la parallelizzazione, non può sapere cosa fa la procedura invocata ad ogni ciclo iterativo e pertanto, conservativamente, effettua il ciclo di do in modo sequenziale. La direttiva di parallelizzazione, in questo caso, è usata a sproposito e con esiti generalmente disastrosi se la subroutine `esegui_questo` non è scritta in modo che le aree di memoria eventualmente modificate siano totalmente diverse per ciascuno dei valori assunti dall'indice k e se si hanno dipendenze dall'ordine di esecuzione ossia se le azioni effettuate da una call dipendono da quelle fatte da una qualsiasi altra call nell'ambito del loop parallelizzato.

Le direttive possono suddividersi in due categorie fondamentali:

- Direttive per ambienti a memoria condivisa (SMP) che pilotano direttamente l'esecuzione specificando quello che il compilatore deve attuare o fornendo informazioni riguardanti dipendenze tra i dati. Sono le direttive di più antica tradizione.
- Direttive per ambienti a memoria sia condivisa che distribuita, che pilotano indirettamente l'esecuzione e che servono ad attribuire i dati ai processori disponibili. I processori si ripartiscono l'attività in modo automatico sulla base dei dati di loro pertinenza che debbono venire modificati. Questa tipologia di direttive è caratteristica dell' High Performance Fortran (HPF) che solo di recente è disponibile al CILEA.

A proposito del primo gruppo di direttive esiste una esigenza di standardizzazione specifica, storicamente scarsa, poiché esse possono considerarsi "discendenti" delle direttive di vettorizzazione nate per favorire l'esecuzione sui primi calcolatori capaci di una forma particolare di calcolo parallelo, quello attuato tramite l'utilizzo di componenti funzionali realizzati all'interno della CPU stessa ed operanti su particolari tipi di registri in modo da attuare in parallelo le sotto_operazioni che costituiscono una operazione elementare ma relativamente complessa quale è la moltiplicazione in virgola mobile.

L'essenza delle direttive proprietarie HP di parallelizzazione è riassunta in questo breve elenco:

	Direttiva	*Descrizione*
D1	`!$dir loop_parallel`	Applicata ai cicli di do.
D2a	`!$dir parallel`	Blocco di sorgente eseguito da vari thread
D2b	`!$dir end_parallel`	Fine del blocco
D3a	`!$dir begin_tasks`	Ogni thread fa un lavoro diverso
D3b	`!$dir next_task`	Lavoro del thread successivo
D3c	`!$dir end_tasks`	Fine dei lavori in parallelo
D4a	`!$dir critical_section`	Blocco di sorgente eseguito da un thread alla volta
D4b	`!$dir end_critical`	Fine del blocco da eseguire non contemporaneamente

Come si vede in sostanza sono previste tre modalità di esecuzione ed è però disponibile un meccanismo che garantisce che un certo lavoro venga svolto non in contemporanea ma da un solo thread alla volta. Questa esigenza si verifica, tipicamente quando ogni thread deve aggiungere in un qualche modo (incluso quello base di addizionare un termine ad una sommatoria) il proprio contributo lavorativo a quello degli altri memorizzandolo in una qualche area di memoria. Per far questo, in modo più o meno esplicito, ricopia localmente l'area di memoria, la modifica e rimette a disposizione di tutti il risultato. Se però, nel frattempo, un altro thread avesse fatto la stessa cosa il risultato non conterrebbe l'aggiornamento di uno dei due thread e questo, verificandosi in modo largamente casuale, produrrebbe errori insidiosi e difficili da individuare poiché, in certe situazioni, a parità di dati iniziali, il risultato finale potrebbe persino essere esatto.

Delle tre modalità di parallelizzazione la prima (D1) è, probabilmente, la più usata e flessibile perché applicabile ai cicli di do che sono ampiamente usati in Fortran, e con il vantaggio di adattarsi a varie disponibilità di processori. La seconda modalità, consistente nel delimitare (D2a, D2b) una porzione di sorgente da far eseguire ai processori attivati ed è analoga alla prima ma non sfrutta una variabile particolare, ovvero l'indice del do, per differenziare il lavoro svolto dai vari thread. Per essere praticamente utilizzabile occorre una funzione ad hoc per riconoscere il numero del thread in azione e operare su dati distinti in base a tale indice. Occorre far dunque uso delle funzioni `my_thread()` e `num_threads()`. Particolare importante: il valore di `my_thread()` è un intero non negativo ed inferiore a `num_threads()`.

La terza modalità (D3a,D3b e D3c) è quella forse concettualmente più semplice ma anche, in un certo senso delicata per l'equilibramento del lavoro tra i vari thread. E' infatti piuttosto difficile, avendo assegnato ad ogni thread un blocco di istruzioni diverso, riuscire a fare in modo che i tempi impiegati per eseguire i vari blocchi siano sostanzialmente analoghi per cui sia ridotto il numero di threads fermi ed inattivi al punto di sincronizzazione rappresentato dalla direttiva D3c.

Le varie tipologie di direttive ammettono di essere accompagnate da una lista di attributi che, benché opzionale, risulta molto spesso indispensabile. Nel Fortran, ad esempio, l'indice del ciclo di do è considerata una variabile privata del thread ossia ogni thread usa in realtà una diversa area di memoria per gestire i valori della propria versione dell'indice. In C, viceversa, occorre esplicitamente individuare questa "primary induction variable". Utile anche, a volte, indicare che si intende non attivare nuovi thread ma sfruttare quelli già attivi per evitare di avere un aggravio eccessivo nelle operazioni di attivazione.

Un aspetto fondamentale per la fattibilità stessa del calcolo parallelo è la dichiarazione di quali variabili, tipicamente scalari, debbano venire clonate in modo che ogni thread disponga di una propria copia privata. Il caso tipico è quello in cui il ciclo di do a cui si è anteposta la direttiva `!$dir loop_parallel` contiene un altro ciclo di do che dunque viene eseguito in parallelo da tutti i thread. L'indice del do interno, uno scalare, in assenza di indicazioni specifiche è una grandezza condivisa ovvero esiste un'unica locazione di memoria a cui hanno accesso tutti i thread; ne consegue che ogni thread incrementerà l'indice del do interno in base alle proprie esigenze e senza il permesso degli altri thread che, da parte loro si comporteranno nello stesso modo con risultati a dir poco catastrofici. La soluzione per uno scalare normale, non indice del do, è quella di trasformare lo scalare stesso in un vettore con un numero di elementi uguale al numero di iterazioni del ciclo di do. Si tratta di una soluzione piuttosto costosa in termini di memoria a cui, evidentemente, è da preferirsi l'uso di una apposita direttiva. La così detta

privatizzazione dei dati si attua con le direttive:

- `!$dir loop_private(…)` applicata ai cicli di do.
- `!$dir parallel_private(…)` applicata alle regioni parallele.
- `!$dir task_private(…)` applicata ai blocchi di istruzione eseguiti in parallelo.

E' consentito l'uso contemporaneo di più direttive nello stesso commento. Ad esempio:

```
!$dir loop_parallel, loop_private(j)
do k=1,4
    v(k)=a(k)
    do j=k+4,n,4
        v(k)=v(k)+a(j)
    end do
end do
```

Si noti che la direttiva `loop_parallel` è applicabile anche nel caso in cui il contatore principale del ciclo iterativo non è immediatamente riconoscibile. In tal caso però occorre indicare esplicitamente al compilatore quale è la variabile indice del ciclo. Ad esempio in questo ciclo di do infinito non c'è un indice di do esplicito, dunque va detto:

```
j=1
!$dir loop_parallel(ivar=j)
mioloop:do
   a(j)=….
   j=j+1
   if(j.le.n) exit mioloop
end do mioloop
```

Un' altra questone importante per la realizzazione di programmi paralleli è quella del dialogo tra procedure diverse nell'ambito di uno stesso ciclo di do parallelo. Per fissare le idee si consideri il seguente esempio:

```
!$dir loop_parallel, loop_private(k,s)
do j=1,n
    s=a(1,j)+b(1,j)
    do k=2,n
        s=a(k,j)+b(k,j)
    end do
   do k=1,n
       c(k,j)=s*a(k,j)
   end do
end do
```

Supponiamo di voler usare subroutine per realizzare le sommatorie e il cambiamento di scala. Questo, nella realtà sarà bene fare perché i calcoli potrebbero essere molto più complessi di quanto mostrato in questo esempio.

Dunque un buon esempio di bella programmazione sarebbe il trasformare il precedente ciclo di do parallelo in questo:

```
!$dir loop_parallel
do j=1,n
    call faseprima(j)
    call faseseconda(j)
end do
```

Supponiamo di avere bisogno di ridurre al minimo il numero di argomenti passato alle subroutine e che le subroutine stesse vedano i dati tramite una qualche use di un qualche modulo chiamato miematrici.
Per non sforzar troppo la fantasia (ma facendo attenzione alle direttive usate) sia dunque:

```
subroutine faseprima(p)
    use miematrici
    integer,intent(in) ::p
    integer::k
    common/ponte/ s
!$dir thread_private(/ponte/)
    s=a(1,p)+b(1,p)
    do k=2,n
        s=a(k,p)+b(k,p)
    end do
    return
end subroutine faseprima
```

ed analogamente per la seconda subroutine:

```
subroutine faseseconda(p)
    use miematrici
    integer,intent(in)::p
    integer::k
    common/ponte/ s
!$dir thread_private(/ponte/)
    do k=1,n
        c(k,p)=s*a(k,p)
    end do
    return
end subroutine faseseconda
```

Senza la direttiva `thread_private` il common labellato `/ponte/` conterrebbe la variabile s condivisa da tutti i thread e dunque non sarebbe possibile far dialogare senza interferenze esterne due subroutine usate dallo stesso thread. L'unico modo sarebbe quello di passare s in argomento il che è fattibilissimo qui ma molto poco pratico in certe situazioni reali.
Questo esempio si presta anche ad un'altra osservazione: le variabili locali interne alle subroutine in parallelo vengono automaticamente allocate dinamicamente e quindi ogni thread ha una copia diversa di k senza bisogno di ricorrere a una qualche direttiva `loop_private(k)`. Si noti che una soluzione al-

ternativa, valida anche per compilatori che non producono automaticamente subroutine "rientranti" è quella che si è adottata nella parallelizzazione tramite moduli: dichiarare `recursive` le due subroutine.
Per concludere questa veloce panoramica delle direttive proprietarie HP consideriamo il problema di introdurre delle sincronizzazioni tra i vari thread.
In linea di principio, considerata la dannosità di tali operazioni dal punto di vista delle prestazioni, ci si dovrebbe impegnare a ridurre al minimo il ricorso a tali direttive.
La direttiva più semplice da usare è la critical_section di cui viene riportato qui un esempio tipico:

```
!$dir loop_parallel, loop_private(varutile)
do j=1,n
    varutile=funmia(x(j))
!$dir critical_section
    somma=somma+varutile
!$dir end_critical_section
end do
```

Poichè nella regione di codice delimitata dalla direttiva `!$dir critical_section` può entrare un solo thread alla volta, la chiamata della function `funmia` dentro la zona della `critical_section` rallenterebbe enormemente il calcolo perché, in pratica potrebbe sempre lavorare un solo thread alla volta. Usando una variabile di servizio, `varutile`, che devo però privatizzare, posso sperare di ottenere ancora prestazioni accettabili se il tempo di calcolo di `funmia(…)` è molto superiore al tempo di accesso ed abbandono della zona della `critical_section`.
Per la sincronizzazione, oltre alle zone critiche, sono disponibili i cancelli e le barriere. Entrambi questi tipi di sincronizzazione richiedono che sia dichiarata una apposita variabile di controllo del singolo cancello o barriera in modo che, in caso di utilizzo di vari cancelli o barriere, ciascuno impostato con propri valori operativi, sia possibile distinguerli.
In sostanza un cancello funziona esattamente come una zona critica ma non è vincolato ad essere applicabile solo se associato alla direttiva `loop_parallel`. La barriera funziona invece in modo diverso ossia è un vincolo che pone in stato di attesa i processi fino a quando il numero di processi voluto ha raggiunto la barriera stessa. Quando i processi bloccati in attesa sono tanti quanto richiesto, vengono liberati tutti contemporaneamente e ciascuno di essi può proseguire l'esecuzione dal punto immediatamente a valle di quello in cui stava bloccato. Si noti che tanto il cancello quanto la barriera sono punti di sincronizzazione "disgiunti" ovvero la stessa barriera può trovarsi in vari punti ed in varie subroutine ossia due processi possono venire bloccati pur trovandosi in subroutine diverse ma contenenti la stessa barriera: quello che identifica il cancello o la barriera è, infatti, la variabile identificativa che deve essere riconosciuta come tale dal compilatore e resa visibile in tutte le subroutine in cui si vuole mettere uno stesso cancello o barriera.
Fortunatamente il processo di standardizzazione è in atto, si è concretizzato con la nascita di un consorzio (OpenMP) che si prefigge lo scopo di formulare uno standard accettato progressivamente da tutte le case costruttrici di calcolatori paralleli a memoria condivisa indipendente dal sistema operativo (Unix o Windows). L'HP si sta progressivamente adeguando alle specifiche che con la versione 2.4 del compilatore F90 sono almeno parzialmente rispettate.Infatti con la versione attualmente installata di F90 l'utente del CILEA ha a disposizione le direttive OpenMP più importanti ovvero, per fare la corrispondenza con l'elenco delle direttive HP essenziali:

	Direttiva	*Descrizione*
D1	`!$omp parallel do`	Applicata ai cicli di do.
D2a	`!$omp parallel`	Blocco di sorgente eseguito da vari thread
D2b	`!$omp end parallel`	Fine del blocco
D3a	`!$omp sections`	Ogni thread fa un lavoro diverso
D3b	`!$omp section`	Lavoro del thread successivo
D3c	`!$omp end sections`	Fine dei lavori in parallelo
D4a	`!$omp critical`	Blocco di sorgente eseguito da un thread alla volta
D4b	`!$omp end critical`	Fine del blocco da eseguire non contemporaneamente

Si può notare come esista una corrispondenza quasi uno a uno tra direttive OpenMP e direttive parallele HP. La conversione di un programma parallelizzato con direttive HP in un programma parallelo portabile su altre piatta-

forme a memoria condivisa dovrebbe risultare perciò abbastanza semplice e, si potrebbe dire, praticamente automatico.
Dev'essere osservato però che, mentre le direttive HP tengono "nativamente conto" delle caratteristiche architetturali delle macchine, l'implementazione delle direttive OpenMP potrebbe risultare più delicata dal punto di vista delle prestazioni per la maggiore generalità del modello di calcolatore per cui tale standard è stato proposto. Potrebbero essere timori infondati: sarebbe però auspicabile non doversi trovare ad un bivio tra il dover privilegiare l'efficienza o la portabilità del programma parallelizzato.
Se siamo inoltre convinti che le innovazioni introdotte nei linguaggi di programmazione negli ultimi venti anni, siano effettivi progressi validi non solo per la programmazione in ambito commerciale ma anche per quella scientifica non possiamo non notare un certo grado di arretratezza nella formulazione della versione 1 dello standard. Per citare la carenza più macroscopica, non viene di fatto ancora recepita l'idea che l'uso dei COMMON labellati sia da scoraggiare in modo altrettanto deciso quanto l'uso dei GOTO del Fortran 77. Dunque è (deo gratias) ammesso l'uso dei moduli del Fortran 90, che consentono di dare una unica definizione di tipi derivati e di caratteristiche della memoria allocata, ma non esiste ancora (si spera ardentemente nell'imminente versione 2) la possibilità di definire moduli contenenti memoria locale, diversa da thread a thread. Come si è visto nella parte dedicata alla parallelizzazione tramite moduli, questi moduli "privati al thread") sono strutture facilmente realizzabili "manualmente" e oltretutto rappresentano una ottima metafora della suddivisione fisica della memoria distribuita ma evidentemente gli estensori della versione 1 delle OpenMP hanno pensato che andasse assecondato l'oscurantismo informatico di una parte degli attuali programmatori scientifici.

Bibliografia

Bastano pochi indirizzi da cui partire per un approfondimento delle questioni trattate in questa panoramica sul calcolo parallelo.
Per quanto concerne le direttive per ambienti Shared-Memory :

- http://www.openmp.org/ ovvero l'home page del consorzio OpenMP che ha come partners, oltre ad HP anche Compaq-Digital, Silicon Graphics, IBM, Intel, SUN…
- http://rodin.cs.uh.edu/wompat2000/Program.html ovvero il programma dell'imminente "Workshop on OpenMP Applications and Tools" tenuto al San Diego Supercomputer Center, San Diego, California il 6-7 luglio 2000 dove si discuterà anche della imminente release 2.0

Per quanto concerne le direttive di "Data Parallelism" si guardi in:

- http://www.crpc.rice.edu/HPFF/ si tratta della home page dell' High Performance Fortran.
- http://www.pgroup.com/prodhpf.htm ovvero la pagina dedicata al compilatore HPF fornito dal "Portrland Group". Si tratta di un precompilatore che traduce nel Fortran disponibile sul sito. Notare che "*PGHPF 3.0* is not yet fully compliant to the ISO/ANSI Fortran 90 standard". Le cose non ancora accettate non sono molte ma … anche questa è la prova di come certe cose evolvano lentamente.

Per quanto concerne il parallelismo a scambio di messaggi:

- http://www-unix.mcs.anl.gov/mpi/ è la pagina ufficiale del consorzio MPI. Il Fortran usabile nelle chiamate alle subroutine MPI è sostanzialmente il 77. Poiché il 90 include il 77 o.k. Volendo comunque usare anche ciò che il 90 dà in più… ci sono trucchi per fare qualcosa ma vanno usati "con le pinze".

Per quando riguarda l'estensione delle CoArray che, anche se ancora, in un certo senso, allo stato "sperimentale" ricordiamo qui per l'interesse teorico della proposta:

- http://www.co-array.org/ ovvero la pagina ufficiale. Le CoArray sono disponibili attualmente sul Cray T3E ed esistono precompilatori che traducono in direttive OpenMP. Dunque al momento la loro diffusione è legata a quella degli ambienti che accettano le OpenMP.

Naturalmente, visto che i supercalcolatori del CILEA sono prodotti dall' HP è opportuno tener d'occhio i loro siti:
http://www.hp.com/esy/lang/fortran/

http://docs.hp.com/dynaweb/hpux11/dtdcen1a/